Statechart Verification with iState

Dai Tri Man Lê

Department of Computer Science,
University of Toronto,
Toronto, ON, M5S 3G4 Canada
ledt@cs.toronto.edu

Abstract. This paper is the long version of the extended abstract with the same name [9]. We describe in detail the algorithm to generate verification conditions from statechart structures implemented in the iState tool. This approach also suggests us a novel method to define a version of predicate semantics for statecharts analogous to how we assign predicate semantics to programming languages.

1 Introduction

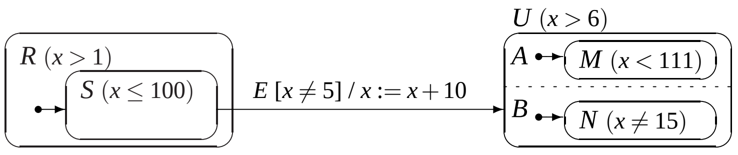
The statechart formalism, proposed by Harel [7] as an extension of conventional finite state machines, is a visual language for specifying reactive systems. It addresses the state explosion problem of state transition diagrams when modeling systems with parallel threads of control by introducing the concepts of *hierarchy*, *concurrency*, and *communication*.

The *iState* tool translates statecharts into various programming languages, currently the Abstract Machine Notation (AMN) of the B method [1], Pascal, and Java. The translation is based on a definition of statecharts in terms of an extension of Dijkstra's guarded commands [15,16]. This work demonstrates a novel statechart verification approach using *state invariants* that has been added to *iState*.

2 Invariants

Statecharts allow executable specifications to be derived from user requirements. We propose to supplement a statechart specification by *invariants*. These are attached to states and specify what has to hold in a state configuration. Invariants are also derived from the requirements. They are not meant for execution, but they allow the statechart specification to be cross-checked. By themselves, statecharts do not lead to opportunities for consistency checks beyond well-formedness; invariants address this limitation and give a way of documenting the "purpose" of states.

Formally, invariants are predicates over global variables, like x in the example below, and states (state tests):



The definition of statecharts in [15,16] translates states into variables and events into (nondeterministic) operations, in which use of the *independent (parallel) composition* of statements is made; the parallel composition operator is essential for translating events with transitions in concurrent states. Using AMN, the states of the previous statechart are translated to variables $root \in \{R,U\}$, $r \in \{S\}$, $a \in \{M\}$ and $b \in \{N\}$ and the event E is translated to:

```
$E \triangleq$ if $root = R$ then
    if $r = S$ then
        if $x \neq 5$ then
            $x := x + 10 \parallel root := U \parallel a := M \parallel b := N$
        end
    end
end
```

Let *si*: *State* $\rightarrow$ *Condition* be a function that assigns to each state the invariant specified by the designer, or *true* if none is specified, together with a test for being in that state. For example:

$$si(S) = (r = S \wedge x \leq 100)$$
$$si(U) = (root = U \wedge x > 6)$$

By the hierarchical structure of statechart, being in a state also means being in all of its ancestor states, in exactly one of its child states if the state is an XOR state, and in all of its child states if the state is an AND state. Hence, we have to compose state invariants together to create the *accumulated invariant* $ai(s)$ of state s. For example:

$$ai(S) = (root = R \wedge x > 1) \wedge (r = S \wedge x \leq 100)$$
$$ai(U) = (root = U \wedge x > 6) \wedge ((a = M \wedge x < 111) \wedge (b = N \wedge x \neq 15))$$

Formally, let *Basic*, *XOR*, *AND* be disjoint subsets of the set *State*. The accumulated invariant *ai*: *State* $\rightarrow$ *Condition* is defined with the help of the *child invariant ci*: *State* $\rightarrow$ *Condition* as follows:

$$ci(s) \triangleq \begin{cases} si(s) \wedge \bigotimes ci[children[\{s\}]] & \text{if } s \in XOR \\ si(s) \wedge \bigwedge ci[children[\{s\}]] & \text{if } s \in AND \\ si(s) & \text{if } s \in Basic \end{cases}$$
$$ai(s) \triangleq \bigwedge si[parent^{+}[\{s\}]] \wedge ci(s)$$

Here, $children[\{s\}]$ denotes the set of all child states of a state s and $parent^{+}[\{s\}]$ denotes the set of all ancestor states of s, where *parent* is the inverse of the *child* relation, $parent = child^{-1}$ [15,16]. The operator $\bigotimes$ stands for *xor*. The definition reflects the meaning of XOR and AND states.

3 Event Codes and Verification Tuples

For each transition $E[guard]/action$ from state S to T, where *action* is a statement that may read and write to global variables, may include state tests, and may broadcast other events, a verification condition is generated:

$$\{ai(S) \wedge guard\}\ action\ \{ai(T)\}$$

In the case of broadcasting in *action*, the broadcast is replaced by a call to the corresponding operation. In the case of transitions in concurrent states on the same event E, a combined transition is considered. In the example, the verification condition for event E is:

$$\begin{array}{l} \{(root = R \wedge x > 1) \wedge (r = S \wedge x \leq 100) \wedge x \neq 5\} \\ \ x := x + 10 \parallel root := U \parallel a := M \parallel b := N \\ \{(root = U \wedge x > 6) \wedge ((a = M \wedge x < 111) \wedge (b = N \wedge x \neq 15))\} \end{array}$$

Hence, our goal is to automate the generating verification condition process. However, before doing so we need to have data structures to store and manipulate the verification conditions efficiently.

Using the algorithm discussed in [15,16], we map each statechart data structure into nondeterministic operations which is represented using an abstract syntax tree (AST). The AST of intermediate language is stored using the data type *EventCode*

$$EventCode \triangleq Identifier \rightarrowtail\!\!\!\!\!\!+ \ Statement$$

where $S \mapsto T$ denotes the set of partial function from S to T.

Let $\mathbb{P}(S)$ and $\mathbf{seq}(S)$ denote the types power set of S and finite sequences of S respectively. Also let $S \leftrightarrow T$ denote the set of relation from S to T. Each *Statement* is then defined as a recursive data type

$$\begin{array}{rl} Statement \triangleq & StateAssign \quad State \\ | & Assignment \quad Identifier\ Expression \\ | & Bcast \quad Identifier \\ | & Guard \quad Condition \leftrightarrow Statement \\ | & Par \quad \mathbf{seq}(Statement) \\ | & Seq \quad \mathbf{seq}(Statement) \\ | & Skip \end{array}$$

where:

- *StateAssign State*: denotes the state assignment node.
- *Assignment Identifier Expression*: denotes assignment node and the left hand side of the assignment is an identifier and the right hand side is an expression.
- *Bcast Identifier*: denotes a broadcasting of an event whose name is represented using an identifier.

- *Guard* $\quad$ *Condition* $\leftrightarrow$ *Statement*: denotes an *alternative choice* where each choice is guarded using a condition. Notice that we use *Condition* $\leftrightarrow$ *Statement* to emphasize the possible non-determinism. When several conditions are true at the same time, a choice is made non-deterministically.
- *Par* $\quad$ **seq**$(Statement)$: denotes the parallel composition of a sequence of statements. Due to the commutativity of parallel composition, we might use a set of statements instead of sequence. However, we decide to use sequence here only for the sake of determinism.
- *Seq* $\quad$ **seq**$(Statement)$: denotes the sequential composition of a *sequence* of statements.
- *Skip* denotes the skip statement.

The type *Condition* can either be a state test or a predicate, which is defined as following

$$\begin{aligned} Condition \triangleq\ & StateTest \quad State \\ \mid\ & Predicate \quad Expression \end{aligned}$$

Notice that using a functional language like syntax to define *Statement* and *Condition* allows us to employ pattern-matching on these data types when presenting our algorithms.

We then generate verification conditions from *EventCode* data structure. This will be done by analyzing the structure of *EventCode* to generate *local verification conditions* and composing them together suitably to produce the final verification conditions. We treat all of these verification conditions uniformly using the notion of *verification tuple*, which we think to be a more suitable representation of Hoare's triple for our verification purpose. The verification tuple type is define as following:

$$VTuple \triangleq \mathbb{P}(State) \times Expression \times Statement \times \mathbb{P}(State)$$

where for each $(s, g, a, t) \in VTuple$ we have:

- s denotes the set of source states of the transition,
- g denotes the guard condition of the transition,
- a denotes the statement which changes the states of global variables (including state variables),
- t denotes the set of target states of the transition.

Notice that each $(s, g, a, t) \in VTuple$ is converted into the following verification condition

$$\{\bigwedge ai[s] \wedge g\}\ a\ \{\bigwedge ai[t]\}$$

which is verified as a normal Hoare's triple. However, using sets of source of target states gives a more optimal way of composing verification conditions together. This also helps us avoiding redundancy when generating accumulating invariants due to states having common ancestors. In other words, we use the following more efficient way to calculate the acumulated invariant of a state set. Let the state set closure function $cl : \mathbb{P}(State) \rightarrow \mathbb{P}(State)$ be defined as following

$$cl(ss) \triangleq \bigcup\{parent^*[\{s\}] \mid s \in ss\}$$

where R^* denotes the reflexive transitive closure of the relation R. We define the set accumulated invariant function $sai : \mathbb{P}(State) \rightarrow Condition$ as follows:

$$\overline{si}(s,ss) \triangleq \begin{cases} si(s) & \text{if } s \in Basic \\ si(s) & \text{if } s \in AND \cup XOR \wedge children(s) \in cl(ss) \\ ci(s) & \text{otherwise} \end{cases}$$

$$sai(ss) \triangleq \bigwedge \{\overline{si}(s,ss) \mid s \in cl(ss)\}$$

For convenience, we let $ast : String \rightarrow Expression \cup Statement$ denote the mapping from a String to its AST. Hence, the verification condition discussed in our example previously can be expresses using the following *VTuple*:

$$(\{R,S\}, ast(\text{“}x \neq 5\text{”}), ast(\text{“}x := x+10 \parallel root := U \parallel a := M \parallel b := N\text{”}), \{U,M,N\})$$

4 Invariant Verification Algorithm

Before presenting the algorithm for generating verification tuples, we need several auxiliary functions. We divide *Condition* into *StateTest* and *Predicate*. The state test condition *StateTest* are generated by EventCode generators using the algorithm in [15,16]. Hence, we let *Predicate* denote the transition guards supplied by users. To distinct these two cases, we use the following function

$$\begin{aligned} &c2tuple \;:\; Condition \rightarrow VTuple \\ &c2tuple \quad StateTest\ s = (\{s\}, true, Skip, \emptyset) \\ &c2tuple \quad Predicate\ e = (\emptyset, e, Skip, \emptyset) \end{aligned}$$

Considering the following example where you have:

$$\textbf{if } c_1 \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ end} \parallel \textbf{if } c_2 \textbf{ then } s_3 \textbf{ else } s_4 \textbf{ end}$$

For simplicity, we suppose these two statements are simply user statements without state test or state assignment. Then the statement $\textbf{if } c_1 \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ end}$ corresponds to the verification tuple set

$$\{(\emptyset, c_1, s_1, \emptyset), (\emptyset, \neg c_1, s_2, \emptyset)\}$$

and the statement $\textbf{if } c_2 \textbf{ then } s_3 \textbf{ else } s_4 \textbf{ end}$ corresponds to the set

$$\{(\emptyset, c_2, s_3, \emptyset), (\emptyset, \neg c_2, s_4, \emptyset)\}$$

Since these two if statements are composed in parallel, we the resulted parallel product verification tuple set for the whole statement is:

$$\{(\emptyset, c_1 \wedge c_2, s_1 \| s_3, \emptyset), (\emptyset, \neg c_1 \wedge c_2, s_2 \| s_3, \emptyset), (\emptyset, \neg c_1 \wedge c_2, s_2 \| s_3, \emptyset), \\ (\emptyset, \neg c_1 \wedge \neg c_2, s_2 \| s_4, \emptyset)\}$$

The more general case including state tests are tackled using the following functions:

$$\begin{array}{ll}
parProd & : \ \mathbf{seq}(\mathbb{P}(VTuple)) \rightarrow \mathbb{P}(VTuple) \\
parProd & [] = \emptyset \\
parProd & [s] = s \\
parProd & [s_1, \ldots, s_n] = \{concat1([t_1, \ldots, t_n]) \mid (t_1, \ldots, t_n) \in s_1 \times \ldots \times s_n\}
\end{array}$$

$$\begin{array}{ll}
concat1 & : \ \mathbf{seq}(VTuple) \rightarrow VTuple \\
concat1 & [] = (\emptyset, true, Skip, \emptyset) \\
concat1 & [(s_i, g_i, a_i, t_i) \mid i = 1..n] = (\bigcup_{i=1}^{n} s_i, \bigwedge_{i=1}^{n} g_i, Par\ [a_1, \ldots, a_n], \bigcup_{i=1}^{n} t_i)
\end{array}$$

This function *parProd* is used very often in our implementation and it helps to simplify the implementation substantially.

Similar to the case of parallel composition is the case sequential composition. For example, an event code

$$\mathbf{if}\ c_1\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{end}\ \ ;\ \ \mathbf{if}\ c_2\ \mathbf{then}\ s_3\ \mathbf{else}\ s_4\ \mathbf{end}$$

will be translated into the following verification tuple set

$$\{(\emptyset, c_1 \wedge c_2, s_1; s_3, \emptyset), (\emptyset, \neg c_1 \wedge c_2, s_2; s_3, \emptyset), (\emptyset, \neg c_1 \wedge c_2, s_2; s_3, \emptyset), \\ (\emptyset, \neg c_1 \wedge \neg c_2, s_2; s_4, \emptyset)\}$$

Since the statements must be composed sequentially, the functions are implemented as following.

$$\begin{array}{ll}
seqProd & : \ \mathbf{seq}(\mathbb{P}(VTuple)) \rightarrow \mathbb{P}(VTuple) \\
seqProd & [] = \emptyset \\
seqProd & [s] = s \\
seqProd & [s_1, \ldots, s_n] = \{concat2([t_1, \ldots, t_n]) \mid (t_1, \ldots, t_n) \in s_1 \times \ldots \times s_n\}
\end{array}$$

$$\begin{array}{ll}
concat2 & : \ \mathbf{seq}(VTuple) \rightarrow VTuple \\
concat2 & [] = (\emptyset, true, Skip, \emptyset) \\
concat2 & [(s_i, g_i, a_i, t_i) \mid i = 1..n] = (\bigcup_{i=1}^{n} s_i, \bigwedge_{i=1}^{n} g_i, Seq\ [a_1, \ldots, a_n], \bigcup_{i=1}^{n} t_i)
\end{array}$$

We then define a function *s2tuples* which maps each statement node to a set of verification tuples. Using pattern-matching, the function *s2tuples* can be defined as following:

$$\begin{array}{ll}
s2tuples & : \ Statement \rightarrow \mathbb{P}(VTuple) \\
s2tuples & StateAssign\ s = \{(\emptyset, rue, StateAssign\ s, \{s\})\} \\
s2tuples & Assignment\ i\ e = \{(\emptyset, true, Assignment\ i\ e, \emptyset)\} \\
s2tuples & Bcast\ i = \{(\emptyset, true, Bcast\ i, \emptyset)\} \\
s2tuples & Guard\ r = \bigcup\{parProd([\{c2tuple(c)\}, s2tuples(s)]) \mid (c, s) \in r\} \\
s2tuples & Par\ ss = parProd([s2tuples(s) \mid s \leftarrow ss]) \\
s2tuples & Seq\ ss = seqProd([s2tuples(s) \mid s \leftarrow ss]) \\
s2tuples & Skip = (\emptyset, true, Skip, \emptyset)
\end{array}$$

Hence, we can construct a global verification tuple map for all events using the following function:

$$\begin{aligned}
&\mathit{vtupleMap} \ : \ \mathit{EventCode} \to (\mathit{Identifier} \to \mathbb{P}(\mathit{VTuple}))\\
&\mathit{vtupleMap}\ \mathit{ec} = \mathit{s2tuples} \circ \mathit{ec}
\end{aligned}$$

where the operation $\circ$ denotes the usual function composition, i.e, $f \circ g(x) \triangleq f(g(x))$.

Since each verification tuple $(s,g,s,t) \in \bigcup \mathit{ran}(\mathit{vtupleMap})$, the action s might still contain event broadcasting. We deal with these event broadcasts similarly to the case of parallel composition of event codes. We first apply topological sorting algorithm [10] on $\mathit{vtupleMap}(\mathit{ec})$ to obtain a sequence

$$\mathit{tss} = [(e_1,s_1),\ldots,(e_n,s_n)] \in \mathbf{seq}(\mathit{Identifier} \times \mathbb{P}(\mathit{VTuple}))$$

such that for each (e_i,s_i), the verification tuple set s_i contains the actions that broadcast only the events in the set $\{e_1,\ldots,e_{i-1}\}$. We can always obtain such a list with the assumption that we don't allow circular broadcasting [3]. We next define a function to collect and filter the boardcasts from the action of a verification tuple as following:

$$\begin{aligned}
&\mathit{collectBcast} \ : \ \mathit{Statement} \to \mathbb{P}(\mathit{Identifier})\\
&\mathit{collectBcast} \quad \mathit{Bcast}\ i = \{i\}\\
&\mathit{collectBcast} \quad \mathit{Guard}\ \{(c_i,s_i) \mid i = 1..n\} = \textstyle\bigcup_{i=1}^{n} \mathit{collectBcast}(s_i)\\
&\mathit{collectBcast} \quad \mathit{Par}\ [s_1,\ldots,s_n] = \textstyle\bigcup_{i=1}^{n} \mathit{collectBcast}(s_i)\\
&\mathit{collectBcast} \quad \mathit{Seq}\ [s_1,\ldots,s_n] = \textstyle\bigcup_{i=1}^{n} \mathit{collectBcast}(s_i)\\
&\mathit{collectBcast} \quad s = \emptyset
\end{aligned}$$

$$\begin{aligned}
&\mathit{filterBcast} \ : \ \mathit{Statement} \to \mathit{Statement}\\
&\mathit{filterBcast} \quad \mathit{Bcast}\ i = \mathit{Skip}\\
&\mathit{filterBcast} \quad \mathit{Guard}\ r = \mathit{Guard}\ \{(c,s) \mid (c,s) \in r \wedge \neg \mathit{isBcast}(s)\}\\
&\mathit{filterBcast} \quad \mathit{Par}\ \mathit{ss} = \mathit{Par}\ [s \mid s \leftarrow \mathit{ss} \wedge \neg \mathit{isBcast}(s)]\\
&\mathit{filterBcast} \quad \mathit{Seq}\ \mathit{ss} = \mathit{Seq}\ [s \mid s \leftarrow \mathit{ss} \wedge \neg \mathit{isBcast}(s)]\\
&\mathit{filterBcast} \quad s = s
\end{aligned}$$

$$\begin{aligned}
&\mathit{isBcast} \ : \ \mathit{Statement} \to \mathit{Boolean}\\
&\mathit{isBcast} \quad \mathit{Bcast}\ \_ \ = \ \mathit{true}\\
&\mathit{isBcast} \quad \_ \ = \ \mathit{false}
\end{aligned}$$

In our real implementation, we filter and collect broadcasted events at the same time for the sake of efficiency. We let $\frown$ denote the sequence concatenation operator. Then the process of translating verification tuples with broadcasting to ones without boardcasting is implemented as the following functions:

```
$\mathit{vtupleNoBcast}\ :\ (\mathit{VTuple} \times \mathbf{seq}(\mathit{Identifier} \times \mathbb{P}(\mathit{VTuple}))) \to \mathbb{P}(\mathit{VTuple})$
$\mathit{vtupleNoBcast}\quad ((s,g,a,t)\ ,\ \mathit{tspre}) =$
        if $(\mathit{collectBcast} = \emptyset)$ then
            $\mathit{parProd}([\{(s,g,\mathit{filterBcast}(a),t)\}]^\frown$
            $[s_i \mid (e_i, s_i) \leftarrow tspre \wedge e_i \in collectBcast(a)])$
    else
        $\{(s, g, a, t)\}$
    end
```

The function *vtupleNoBcast* takes a pair of verification tuples and *tspre* as inputs. The argument *tspre* represents the set of prefix of *tss* where all boardcasting are already expanded. Since we *tss* is already topologically sorted according to the dependencies of boardcasting, the action *a* only broadcasts the events defined in *tspre*. Hence, we apply the function *parProd* on the list consisting of the input verification tuples with all the broadcasts filtered and the part of *tspre* chosen according the set of events that the action *a* broadcasts.

The rest of the elimination of broadcasting is defined in the next two functions. We define a function *vtupleSetNoBcast* which is similar to *vtupleNoBcast* but apply on verification tuple sets instead.

$$
\begin{aligned}
&\mathit{vtupleSetNoBcast} : (\mathbb{P}(\mathit{VTuple}) \times \mathbf{seq}(\mathit{Identifier} \times \mathbb{P}(\mathit{VTuple}))) \rightarrow \mathbb{P}(\mathit{VTuple}) \\
&\mathit{vtupleSetNoBcast} \quad (\{v_i \mid i = 1..n\}\ ,\ \mathit{tspre}) \ = \ \textstyle\bigcup_{i=1}^{n} \mathit{vtupleNoBcast}(v_i, \mathit{tspre})
\end{aligned}
$$

We then define the desired function *vseqNoBcast* which can now be applied to the topological sort list *tss* to expand the event broadcasting to suitable verification tuple sets. This function is defined as following:

$$
\begin{aligned}
&\mathit{vseqNoBcast} : \mathbf{seq}(\mathit{Identifier} \times \mathbb{P}(\mathit{VTuple})) \rightarrow \mathbf{seq}(\mathit{Identifier} \times \mathbb{P}(\mathit{VTuple})) \\
&\mathit{vseqNoBcast} \quad [\,] \ = \ [\,] \\
&\mathit{vseqNoBcast} \quad [(e_i, s_i) \mid i = 1..n] \ = \ \mathit{tspre}^\frown[(e_n, \mathit{vtupleSetNoBcast}(s_n, \mathit{tspre}))] \\
&\qquad \mathbf{where} \quad \mathit{tspre} \ = \ \mathit{vseqNoBcast}([(e_i, s_i) \mid i = 1..n-1])
\end{aligned}
$$

Hence, we use the result $\mathit{vseqNoBcast}(\mathit{tss})$ to generate the verification conditions.

5 Predicate Semantics of Statecharts

Our verification approach reveals a strong connection between a statechart transition and a verification tuple. This motivates us to provide a predicate semantic of statecharts instead of the traditional operational way in [8,13,11]. We do so by introducing the functions for translating statecharts to verification tuples. These functions are very similar to the functions used in the previous section. In fact, we will use some auxiliary functions defined previously.

We first need to provide the data structure used to represent statecharts.

$$
\begin{aligned}
\mathit{State} \triangleq\ & \mathit{Basic} \quad \mathit{Identifier} \\
\mid\ & \mathit{And} \quad \mathit{Identifier} \times \mathbf{seq}(\mathit{State}) \\
\mid\ & \mathit{Xor} \quad \mathit{Identifier} \times \mathbf{seq}(\mathit{State}) \times \mathit{State} \times \mathit{Transition}
\end{aligned}
$$

where *Transition* is a five-ary relation defined as following:

$$
\mathit{Transition} \triangleq \mathit{State} \times \mathit{Identifier} \times \mathit{Condition} \times \mathit{Statement} \times \mathit{State}
$$

This definition says a statechart state can be either

- a Basic state with a state name,
- a composite And state encoded by a state name and a sequence of parallel sub states, or
- a composite Xor state encoded by a state name, a sequence of sub states, an initial state and a transition relation.

Each transition is defined by its source state, triggering event, transition guard, action and target state respectively.

We also use an event-centric approach by dealing with each event separately. Hence, we first define a function to return the restriction of a statechart with respect to a specific event. We use $\langle\rangle$ to denote the empty identifier which indicate the "event name" of a spontaneous transitions.

$$\begin{aligned}
&\mathit{resStateEvent} : (\mathit{Identifier} \times \mathit{Sate}) \rightarrow \mathit{State}\\
&\mathit{resStateEvent}\quad (\_\,,\ \mathit{Basic\ id\ s}) = \mathit{Basic\ id\ s}\\
&\mathit{resStateEvent}\quad (e\ ,\ \mathit{And\ id}\ [s_i \mid i = 1..n]) = \mathit{And\ id}\ [\mathit{resStateEvent}(i, s_i) \mid i = 1..n]\\
&\mathit{resStateEvent}\quad (e\ ,\ \mathit{Xor\ \ id\ \ seqs\ \ init\ \ t})\\
&\qquad = \mathit{Xor\ \ id\ \ seqs\ \ init}\ [(ss, e', c, a, ts) \mid (ss, e', c, a, ts) \leftarrow t \wedge (e' = e \vee e' = \langle\rangle)]
\end{aligned}$$

We then need a function to return the verification tuple correspondent to the initialization of a state due to the fact that each composite XOR state must have an initial state.

$$\begin{aligned}
&\mathit{initialize} : \mathit{State} \rightarrow \mathbb{P}(\mathit{Vtuple})\\
&\mathit{initialize}\quad s = \ \mathbf{case}\ s\ \mathbf{of}\\
&\quad \mathit{Basic\ \ id\ \ s} \rightarrow \{(\emptyset, \mathit{true}, \mathit{StateAssign}\ s, \{s\})\}\\
&\quad \mathit{And\ \ id}\ \ [s_i \mid i = 1..n] \rightarrow\\
&\qquad \mathit{parProd}([\mathit{initialize}(s_i) \mid i = 1..n]^\frown[\{(\emptyset, \mathit{true}, \mathit{StateAssign}\ s, \{s\})\}])\\
&\quad \mathit{Xor\ \ id}\ \ [s_i \mid i = 1..n]\ \ \mathit{init\ \ t} \rightarrow\\
&\qquad \mathit{parProd}([\mathit{initialize}(\mathit{init})]^\frown[\{(\emptyset, \mathit{true}, \mathit{StateAssign}\ s, \{s\})\}])
\end{aligned}$$

This function *initialize* will always return a *singleton set*, since we enforce only one possibility to initialize a composite or basic state by disallowing some statecharts variants [15,16]. However, for convenience the returned type of this function is $\mathbb{P}(\mathit{Vtuple})$ to make it easier for composing verification tuples together in the intermediate composition steps. In other words, it allows us to treat this case as a special case of parallel composition using the *parProd* function.

The next step is to define a recursive function to generate verification tuples with respect to statechart data structure. The base case is the the case of basic states and the recursive cases deal with composite states. The most difficult problem is caused by the existence of spontaneous transitions in composite XOR states. We define the following two functions that take a child state of an XOR state and generate verification conditions. When there are spontaneous transitions, the function *getNext* invokes *getSpon* to search for the spontaneous transition going from the target states and generate verification condition for them and then compose the verification conditions together properly.

```
getNext : $(State \times Transition) \rightarrow \mathbb{P}(Vtuple)$
getNext   $(s, tr)$
      $= \{parProd([\{(\{s\}, g, a, \emptyset)\}, getSpon(t, tr)]) \mid (s, id, g, a, t) \in tr \wedge id \neq \langle\rangle\}$

getSpon : $(State \times Transition) \rightarrow \mathbb{P}(Vtuple)$
getSpon   $(s, tr) = S \cup parProd([\{(\emptyset, G, Skip, \emptyset)\}, initialize(s)])$
      **where**
        $S = \{parProd([\{(\emptyset, g, a, \emptyset)\}, getSpon(t, tr)]) \mid (s, id, g, a, t) \in tr \wedge id = \langle\rangle\}$
        $G = \bigwedge\{\neg g \mid (s, id, g, \_, \_) \in tr \wedge id = \langle\rangle\}$
```

The set S in function *getSpon* corresponds to the case the spontaneous transitions are taken and the condition G is the condition for non of the spontaneous transition from state s is taken.

After having these two functions, the rest of the task of generating verification tuples from a statechart state *restricted to one event* is defined in the following function.

```
s2tuples : $State \rightarrow \mathbb{P}(Vtuple)$
s2tuples   $s =$ **case** $s$ **of**
    $Basic\ id\ s \rightarrow \{(\{s\}, true, Skip, \emptyset)\}$
    $And\ id\ [s_i \mid i = 1..n] \rightarrow parProd([s2tuples(s_i) \mid i = 1..n])$
    $Xor\ id\ [s_i \mid i = 1..n]\ init\ tr \rightarrow S_1 \cup S_2$
        **where**
          $S_1 = \bigcup_{i=1}^{n} getNext(s_i, tr)$
          $G = \bigwedge\{\neg g \mid (\_, g, \_, \_) \in S\}$
          $S_2 = parProd([\{(\emptyset, G, Skip, \emptyset)\}]^\frown[s2tuples(s_i) \mid i = 1..n])$
```

To generate all the verification conditions of a statechart, we first collect the set of all events in a given statechart. For each event we use the function *resStateEvent* to get the restricted statechart to each event in the set and then apply the function *s2tuples* to the root of the statecharts. As a result of this process, we will get a map from event names to verification tuples exactly like the map *vtupleMap* previously. The branch statements and event broadcasting in the actions of verification tuples can be easily expanded out using the function *s2tuples* and *vseqNoBcast* defined in the last section.

6 Implementation

The *iState* tool currently uses the Simplify theorem prover [6] to discharge the generated verification conditions because of its support of first order logic and linear arithmetic. Simplify also has arrays built in, though currently *iState* does not use them. We are working on extending *iState* with data types like arrays, rational numbers, and real numbers. In future, we also plan to extend the verification theory to timed transitions [14].

7 Discussion

Compared to the statechart verification approaches in [4,5,12], we use an *event-centric* semantics of statecharts by looking at events as *operations* rather than *data* as in the original *state-centric* semantics [8]. Instead of writing global temporal specification (say in CTL or LTL) separately, inspired by *nested invariant diagram* [2], invariants (*safety properties*) are attached to states.

By attaching invariants to states and utilizing the guarded command representation of statecharts [15,16], we arrive at a rather straightforward verification method. The approach generating verification conditions leads to many small “local” verification conditions and avoids some impossible configurations, compared to when specifying invariants on the global level. As many small verification conditions are easier to handle automatically than a few large ones, we believe that the approach can more easily scale up for the verification of large systems.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
2. R. Back. Invariant based programming revisited. Technical Report 661, TUCS - Turku Centre for Computer Science, Turku, Finland, 2005.
3. M. von der Beck. A comparison of statechart variants. In H. Langmaack, W.-P. deRoever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 863, pages 128–148. Springer-Verlag, 1994.
4. P. Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *ArXiv Computer Science e-prints*, July 2004.
5. E. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical Report CMU-CS-00-XXX, School of Computer Science, Carnegie Mellon University, August 2000.
6. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
8. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(5):293–333, 1996.
9. D.T.M. Le, E. Sekerinski, and S. West, Statechart Verification with iState, FM 06, Canada, 2006.
10. C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.
11. Andrea Maggiolo-Schettini, Adriano Peron, and Simone Tini. A comparison of statecharts step semantics. *Theor. Comput. Sci.*, 290(1):465–498, 2003.
12. E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in PROMELA/SPIN. WIFT, Los Alamitos, CA, USA, 1998. IEEE Computer Society.
13. Shengchao Qin and Wei-Ngan Chin. Mapping statecharts to verilog for hardware/software co-specification. In *FME*, pages 282–300, 2003.
14. S. Samet. Timed transitions in statecharts, from formalization to translation and code. Master’s thesis, McMaster University, Computing and Software Department, 2005.
15. E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001*, LNCS, pages 376–390, Toronto, Canada, 2001. Springer-Verlag.
16. E. Sekerinski and R. Zurob. Translating statecharts to B. In M. J. Butler, L. Petre, and K. Sere, editors, *IFM 2002*, LNCS, pages 128–144, Turku, Finland, 2002. Springer-Verlag.